Last List of Good Problems to Review (Midterm 2 Onward)

```
18.4, 19.3, 19.8 (flow)
22.1,3 (coin method and potential method)
25.1,2,3,4 (get comfortable with probability)
25.9,10 (Slightly more complicated probability analysis)
25.13,19
27.3
```

The Essentials (Compressed List of Problems)

```
1.9, 2.2, 3.1, 4.5, 5.1, 6.1, 7.1, 8.2, 9.8, 10.1, 11.3,
12.6, 13.3, 16.4, 17.3, 18.4, 19.3, 22.1, 25.9, 25.10, 25.12, 27.3
```

25. ***Racetrack*** (also known as *Graph Racers* and *Vector Rally*) is a two-player paper-and-pencil racing game that Jeff played on the bus in 5th grade.[17] The game is played with a track drawn on a sheet of graph paper. The players alternately choose a sequence of grid points that represent the motion of a car around the track, subject to certain constraints explained below.

velocity	position
$(0,0)$	$(1,5)$
$(1,0)$	$(2,5)$
$(2,-1)$	$(4,4)$
$(3,0)$	$(7,4)$
$(2,1)$	$(9,5)$
$(1,2)$	$(10,7)$
$(0,3)$	$(10,10)$
$(-1,4)$	$(9,14)$
$(0,3)$	$(9,17)$
$(1,2)$	$(10,19)$
$(2,2)$	$(12,21)$
$(2,1)$	$(14,22)$
$(2,0)$	$(16,22)$
$(1,-1)$	$(17,21)$
$(2,-1)$	$(19,20)$
$(3,0)$	$(22,20)$
$(3,1)$	$(25,21)$

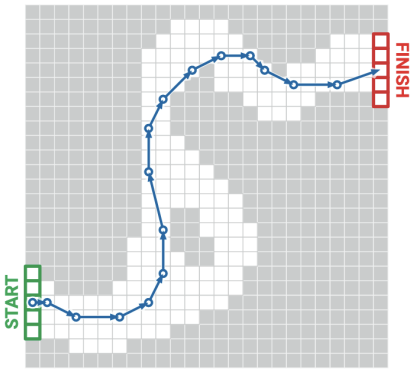


Figure 5.22. A 16-step Racetrack run, on a 25 × 25 track. This is *not* the shortest run on this track.

Each car has a *position* and a *velocity*, both with integer x- and y-coordinates. A subset of grid squares is marked as the *starting area*, and another subset is marked as the *finishing area*. The initial position of each car is chosen by the player somewhere in the starting area; the initial velocity of each car is always $(0,0)$. At each step, the player optionally increments or decrements either or both coordinates of the car's velocity; in other words, each component of the velocity can change by at most 1 in a single step. The car's new position is then determined by adding the new velocity to the car's previous position. The new position must be inside the track; otherwise, the car crashes and that player loses the race. The race ends when the first car reaches a position inside the finishing area.

Suppose the racetrack is represented by an $n \times n$ array of bits, where each 0 bit represents a grid point inside the track, each 1 bit represents a grid point outside the track, the "starting area" is the first column, and the "finishing area" is the last column.

Describe and analyze an algorithm to find the minimum number of steps required to move a car from the starting line to the finish line of a given racetrack.

Exercise 9.15. Here we consider a special case of SAT that we call *ordered-k-overlapping-3-SAT* for a fixed parameter k. We take as input a 3-CNF formula $f(x_1, \ldots, x_n)$ with m clauses listed in order as to satisfy the following property. For every index i, there are at most k variables x_j with index $j < i$ that share a clause with the any variable x_j with index $j \geq i$. For this special case of 3-SAT, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Exercise 12.12. Consider a directed graph $G = (V, E)$ that is *transitively closed*. That is, a vertex u can reach a vertex v iff there is an edge from u to v. Consider the problem of finding a Hamiltonian path in a transitively closed graph. That is, the input consists of a transitively closed graph, and the goal is to find a Hamiltonian path in the graph. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

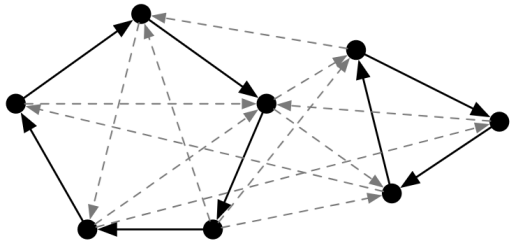
Exercise 12.15. It's-a me, Mario! This problem is inspired by Super Mario World, where Mario must make it from some starting point to the flag in front of a castle, collecting as many coins as possible along the way.

Let $G = (V, E)$ be a directed graph where each vertex $v \in V$ has $C_v \geq 0$ coins. For a walk in G, we say that the *number of coins collected by the walk* is the total sum of C_v over all *distinct* vertices v in the walk. (If we visit a vertex v more than once, we still only get C_v coins total.) Given $s, t \in V$, the goal is to compute the maximum number of coins collected by any (s, t)-walk.

1. (5 points) Let G be a DAG. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

2. (5 points) Let G be a general directed graph. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Exercise 19.10. A *cycle cover* of a given directed graph $G = (V, E)$ is a set of vertex-disjoint cycles that cover every vertex in G. You can also think of a cycle cover as a subset of edges $F \subseteq E$ such that each vertex v has exactly one edge in F entering v, and exactly one edge in F leaving v. This implies that F has exactly n edges. Below is a directed graph where the solid arcs form a cycle cover, and the dashed arcs are the remaining edges in G.



Consider the problem of deciding if a directed graph G contains a cycle cover. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Exercise 25.13. Last week you decided to clean up your toolshed, but you compulsively made the following silly mistake. Originally you had n matching pairs of nuts and bolts of different sizes, but you decided to organize the nuts and bolts separately into a box full of nuts and a box full of bolts, splitting up all the pairs of nuts and bolts in the process. So now you have n nuts, and n bolts, such that each nut fits a distinct bolt and vice-versa, but you don't know which nut goes with which bolt. The bolts all look pretty similar, so you can't compare two bolts directly with each other and tell which one is bigger. Likewise you cannot compare the nuts to each other. However, you can try to fit one nut to one bolt, and see if either:

1. The nut fits the bolt.
2. The nut is too big for the bolt.
3. The nut is too small for the bolt.

We will treat one of these nut-to-bolt tests as a single operation. Your goal is to match up all nuts and bolts. Of course you can compare every pair of nut and bolt in $O(n^2)$ time, but can you do better?

Design and analyze an algorithm, as fast as possible, to recover all matching pairs of nuts and bolts.